

Efficient Distributed Quantum Computing

Robert Beals¹, Stephen Brierley^{2*}, Oliver Gray², Aram Harrow³,
Samuel Kutin¹, Noah Linden⁴, Dan Shepherd^{2,5} and Mark Stather⁵

¹*Center for Communications Research, 805 Bunn Drive,
Princeton, NJ 08540, USA*

²*Heilbronn Institute for Mathematical Research, Dept. of Mathematics,
University of Bristol, Bristol BS8 1TW, UK*

³*Dept. of Computer Science & Engineering, University of Washington,
Seattle, WA 98195, USA*

⁴*Dept. of Mathematics, University of Bristol, Bristol BS8 1TW, UK*

⁵*CESG, Hubble Road, Cheltenham, GL51 0EX, UK*

July 11, 2012

Abstract

We provide algorithms for efficiently addressing quantum memory in parallel. These imply that the standard circuit model can be simulated with low overhead by the more realistic model of a distributed quantum computer. As a result, the circuit model can be used by algorithm designers without worrying whether the underlying architecture supports the connectivity of the circuit. In addition, we apply our results to existing memory intensive quantum algorithms. We present a parallel quantum search algorithm and improve the time-space trade-off for the Element Distinctness and Collision problems.

1 Introduction

There is a significant gap between the usual theoretical formulation of quantum algorithms and the way that quantum computers are likely to be implemented. Descriptions of quantum algorithms are often given in one of two theoretical models: the quantum Random Access Memory (RAM) model (possibly equipped with an oracle) or the circuit model, both of which essentially ignore locality issues. On the other hand, any implementation is likely to be mostly local in two or three dimensions, with a small number of long-range connections and, due to the presumed requirements of fault-tolerance [1], to involve concurrent execution of one- and two-qubit gates with fast classical control.

*electronic address: steve.brierley@bristol.ac.uk

This is unlike classical computers, whose implementations are currently dominated by von Neumann architectures where $O(1)$ cores share access to a large RAM.

We address problems with both theoretical models of quantum computing, demonstrating that a single idea—*reversible sorting networks*—can be used to efficiently relate them to a model of computation that is more physically realistic. For example, we show how quantum computers can efficiently access memory in a parallel and unrestricted way. In fact, the memory does not need to be in a single place, but could be distributed amongst many processors. These ideas are naturally extended to relate any quantum algorithm presented in terms of a quantum circuit to a distributed quantum computer in which each processor acts on a few well-functioning qubits connected to a small number of other memory sites (possibly via long-range interactions). Hence experiments such as NV centres in diamond, and trapped ions connected using optical cavities [12, 24], or cavity QED for superconducting qubit networks [6, 28], could be used to efficiently implement quantum algorithms presented in the circuit model.

Our results can be summarized as relating the following three models of quantum computation presented pictorially in Fig. 1: the well known *circuit model* (where a single processor can perform up to N concurrent operations on any of the qubits), *quantum parallel RAM* (where N processors can each perform one operation per time step on any part of the memory), and a more physically realistic model, *distributed quantum computing* (where processors with local memory are laid out in a certain fixed topology).

We provide an algorithm (circuit) that can look up multiple memory entries in parallel, and prove that this algorithm (measured in terms of circuit complexity) is scarcely more expensive than *any* circuit capable of accurately accessing even a single entry. These results are explained in more detail in §2 and §3 and culminate in the following two Theorems.

Theorem 1. *A quantum circuit which accesses one of N memory bits given its index requires width $\Omega(N)$ and depth $\Omega(\log(N))$.*

Theorem 2. *There is a uniform family of quantum circuits computing, from N indices j_1, \dots, j_N and N bits x_1, \dots, x_N , the N bits x_{j_1}, \dots, x_{j_N} . This circuit family has width $O(N \log N)$ and depth $O(\log N \log \log N)$.*

The word “compute” in Theorem 2 refers to replacing some target registers y_1, \dots, y_N with $y_1 \oplus x_{j_1}, \dots, y_N \oplus x_{j_N}$. The input y_i can be interpreted as the register state of processor i and the computation $y_i \oplus x_{j_i}$ as a memory request by this processor. Therefore, Theorem 2 allows N processors unrestricted access to a shared memory, demonstrating the equivalence (up to log factors) between the quantum parallel RAM and the circuit models.

We also provide an algorithm for efficiently moving data (Theorem 3). Distinct indices are simply permuted, replacing x_1, \dots, x_N with x_{j_1}, \dots, x_{j_N} . Armed with Theorem 3, we relate the circuit model to the distributed quantum computing model in §4. We consider a particular topology that is based on an efficient sorting network, and show that it leads to a physically realistic distributed quantum computer that has a very small

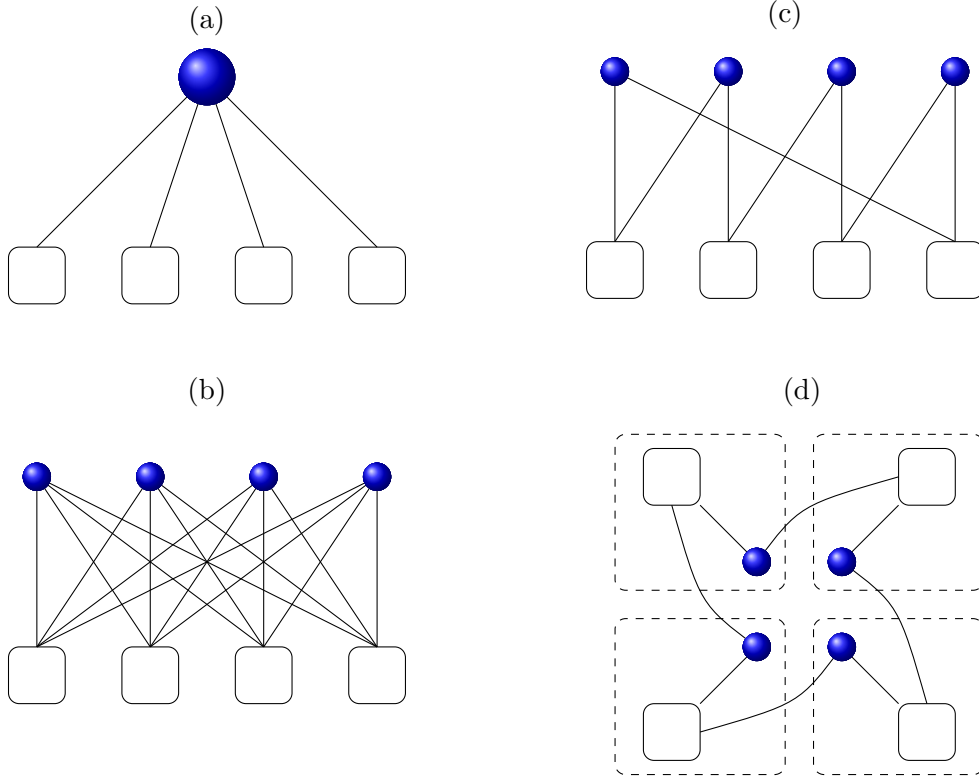


Figure 1: Our results can be summarized as relating the following models of quantum computation: (a) The circuit model – a single processor can access all of the memory and perform up to N operations concurrently; (b) Quantum parallel RAM – N processors with unrestricted access to the memory, each processor can only perform one operation per time step; (c) Distributed quantum computing – N single operation processors with restricted access to the memory and (d) a redrawing of the picture (c) where we emphasize the locality of the model. In the illustration, we depict the case $N = 4$.

overhead for simulating arbitrary quantum circuits.

In §4.2, we take the concept of emulating arbitrary circuits on a restricted, non-monolithic quantum computer one step further. We prove that given (by an experimentalist) *any* layout of qubits grouped into memory sites, a device with this layout can implement algorithms presented in the circuit or quantum RAM models. Of course there is a price to pay: the overhead depends on the topology of the processors, but our algorithm is close to optimal. More precisely, we prove the following Theorem.

Theorem 5. *Let \mathcal{G} be a graph on N vertices, and let $D_{\mathcal{G}}$ denote the minimum depth of a sorting algorithm over \mathcal{G} . Arbitrary quantum circuits of width N can be emulated on \mathcal{G} with an overhead factor of $\tilde{O}(D_{\mathcal{G}} \log N)$. On the other hand, any algorithm for emulating arbitrary circuits must cost at least $\tilde{\Omega}(D_{\mathcal{G}} / \log N)$.*

Thus, the quantum parallel RAM model can be efficiently simulated using a distributed quantum computer. We consider, in §5, various quantum search problems in the quantum parallel RAM model. These include the element distinctness problem, which in the past has been considered in the quantum RAM model [4, 10] as well as in a parallel model with no communication [17]. In both of these models, the best time space trade-off that has been achieved is $ST^2 = \tilde{O}(N^2)$, where S represents memory space, T represents time, and N is the number of elements to test for distinctness. These elements are given by a subroutine or circuit which computes them from their indices. Grover and Rudolph [17] pose beating this trade-off as a challenge which we answer in the following theorem.

Theorem 7. *There is a quantum algorithm solving the element distinctness problem that has a time-space trade-off*

$$ST = \tilde{O}(N). \tag{1}$$

2 The cost of accessing quantum memory

In this Section, we motivate and introduce the primitives of *parallel memory look-ups* and *data-moving*. We describe methods of measuring the *cost* of an algorithm, and give the precise statement of Theorems 1, 2 and 3.

The cost of our algorithms are all specified in terms of the circuit model. All circuits developed in this Section and in the proofs will be comprised of reversible (unitary) gates that map computational basis vectors to computational basis vectors. In this sense, our algorithms in Sections 2, 3 and 4 are entirely *classically reversible*, nowhere introducing nor exploiting quantum superposition. Moreover, all our circuits perfectly clean any ancilla qubits that they use, so that these circuits can be used as quantum subroutines.

Requirements will be shown in terms of *logical unitaries* acting on *input registers*, without always explicitly showing *auxiliary registers*, though such ancillas will naturally be required. These are always presumed to be provided in the canonical pure computational basis state $|0\rangle$, and will always be returned to that state on algorithm termination, independent of any input. In §3, we will prove the Theorems by providing circuits that implement the required logical unitaries.

A circuit is decomposed (perhaps recursively) into a series of subroutines. In the circuit model, every subroutine is built out of gates from a universal gate set (cf. for example, reference [21]). Gates can be implemented concurrently—that is, within the same *timeslice*—whenever they act on disjoint sets of qubits. The *cost* of a circuit is measured by three parameters:

- *depth*, the number of timeslices required in the circuit;
- *size*, the total number of gates in the circuit, and

- *width*, the total number of qubits (inputs + ancillas) required in the circuit.

These three are all taken to be functions of the *logical unitary width*, which is the number of input qubits required for the logical unitary.

2.1 The cost of a single look-up

Often when quantum algorithms are quoted in the *query model*, the concept of an *oracle* is used to abstract away those logical unitaries that are intended to make ‘random’ access to (quantum) memory. In this Section, we examine the idea of accessing memory in more detail, without using oracles.

Definition 1. *A logical unitary for accessing a single piece of data is a map $U_{(1,N)}$ that implements*

$$U_{(1,N)} : |j\rangle|y\rangle|x_1, x_2, \dots, x_N\rangle \mapsto |j\rangle|y \oplus x_j\rangle|x_1, x_2, \dots, x_N\rangle, \quad (2)$$

where \oplus denotes bitwise addition.

Here we have depicted $2 + N$ registers. The first register (*index register*) holds an index large enough to ‘point’ to any one of N *data registers*; its associated Hilbert space must be of dimension at least N . The second register is called the *target register* and holds the same kind of data as a data register. The other N registers are data registers and could in principle be of any (equal) size.

We can derive a simple lower bound for the cost of accessing a single piece of memory based on two simple constraints on any circuit for $U_{(1,N)}$. The circuit must hold the entire database and there must be a causal chain from every data register to the target register. More precisely, we have the following Theorem.

Theorem 1. *If a circuit implements $U_{(1,N)}$ on N data registers each consisting of d qubits, then its width is $\Omega(Nd)$ and its depth is $\Omega(\log(N))$.*

Proof : The width of the circuit must be $\Omega(Nd)$ because this is the logical width of the unitary (the number of input/output qubits). Since any data register could affect the target register, there must be a causal chain of gates from any data register to the target register. Each permitted gate touches $O(1)$ registers, so the depth of the longest chain must be $\Omega(\log(N))$. \square

There is a sense in which the gates in a typical circuit for $U_{(1,N)}$ can be said to be “not working very hard” (although this idea is hard to quantify precisely), and this inefficiency points to the need for a parallel algorithm.

2.2 The cost of parallel look-ups

Definition 2. A logical unitary for accessing N pieces of data is a map $U_{(N,N)}$ that implements

$$\begin{aligned} U_{(N,N)} : |j_1, j_2, \dots, j_N\rangle |y_1, y_2, \dots, y_N\rangle |x_1, x_2, \dots, x_N\rangle \\ \mapsto |j_1, j_2, \dots, j_N\rangle |y_1 \oplus x_{j_1}, y_2 \oplus x_{j_2}, \dots, y_N \oplus x_{j_N}\rangle |x_1, x_2, \dots, x_N\rangle. \end{aligned} \quad (3)$$

Here we have depicted N index registers, N target registers, and N data registers, comprising a total of $3N$ input registers. As before, the index registers are each made up of $\lceil \log_2 N \rceil$ qubits, while the target registers and data registers are each made up of d qubits.

Theorem 2. For data registers of d qubits each, there exists a uniform family of circuits implementing $U_{(N,N)}$, having width $O(N(\log(N) + d))$ and depth $O(\log N \log(d \log N))$.

Proved in §3.4. □

Note that the width of the circuit for $U_{(N,N)}$ is linear in the width of the logical unitary itself, which is the best that could be hoped for. Theorems 1 and 2 tell us that for parallel memory lookups, we can achieve a factor N more ‘effect’ for only a small additional ‘effort’. This will be seen to have radical effects on certain ‘memory-intensive’ algorithms in §5.

2.3 Data-moving

Two qubits can easily be moved using a SWAP gate (for example), but then the locations of the qubits to be exchanged must be known at the time the circuit is created (‘*compile time*’). Moreover, in architectures such as distributed quantum computing (which we will discuss in more detail in §4), it is not possible to implement a SWAP gate between arbitrary qubits, due to the locality constraints. We could use two $U_{(N,N)}$ circuits to perform an arbitrary permutation of the indices, mapping the state $|x_1, x_2, \dots, x_N\rangle$ to $|x_{j_1}, x_{j_2}, \dots, x_{j_N}\rangle$. However, by considering the permutation problem directly, we present a circuit that is simpler and cheaper (by a factor 2) than using $U_{(N,N)}$ twice.

The *No-Cloning Theorem* [29] prevents us from duplicating quantum data when it is not in a known basis, and so a data-moving algorithm is only to be concerned with *permuting* a set of qubits, without trying to copy any. For this reason, the logical unitary for moving data is only defined on states whereby, in every branch of the quantum superposition, the index registers collectively hold *distinct* pointers.

Definition 3. A logical unitary for data moving is a map V_N acting on quantum registers that implements

$$\begin{aligned} V_N : |j_1, j_2, \dots, j_N\rangle |x_1, x_2, \dots, x_N\rangle \\ \mapsto |j_1, j_2, \dots, j_N\rangle |x_{j_1}, x_{j_2}, \dots, x_{j_N}\rangle, \end{aligned} \quad (4)$$

whenever the j_i are all distinct.

As before, the N index registers are each taken to be of $\lceil \log_2 N \rceil$ qubits, while the N data registers are each of d qubits. (It is not important to us how V_N behaves on input states violating the given guarantee, in the same way that we don't care what a circuit does when its ancilla qubits are input in a state other than $|0\rangle$.)

In §3.2, we present quantum circuits for V_N that are closely related to the circuits for $U_{(N,N)}$. (In fact, it is simpler to describe our solution for V_N first.)

Theorem 3. *For data registers of d qubits each, there exists a uniform family of circuits implementing V_N deterministically, having width $O(N(\log(N) + d))$ and depth $O(\log N \log(d \log N))$.*

Proved in §3.4.

□

3 Algorithm Descriptions

In this Section, we describe and analyse algorithms (circuits) for accessing memory in parallel, $U_{(N,N)}$, and data moving, V_N , and prove the corresponding Theorems 2 and 3. These algorithms share the same overall structure. A sorting network is used to sort the data whilst saving some auxiliary bits. Next, we apply a transformation to the sorted data; either a permutation, in the case of data moving, or a general copying transformation, for parallel look-ups. Finally we pass the transformed data through the sorting network in reverse, using the auxiliary bits to ensure that the sorting is correctly reversed.

We begin this Section by describing the main subroutine for the algorithms; a classical reversible sorting network. Then we present the conceptually simpler algorithm for data moving, before going on to describe our parallel look-up algorithm. Both the parallel look-up algorithm and the data-moving algorithm are in fact *classical reversible circuits*. Indeed, the tools we use (principally sorting networks) were developed originally for classical parallel computing [26, 27, 19]. We were unable to find our Theorems 2 or 3 in the literature (although [26] is somewhat similar), and we suspect that may be because quantum computing presents architectural issues that are different to those encountered in the field of traditional classical parallel computing.

3.1 Sorting networks

A *sorting network* is a network on T wires in which each wire represents one of T elements and where the only gates are *binary comparators*. A binary comparator takes two elements on its input, and it outputs the same two elements but in the correct order, according to some specified comparison routine. To make it reversible, each comparator additionally has its own *sorting-ancilla* bit that is to enter the comparator in state $|0\rangle$.

This sorting-ancilla gets flipped to a $|1\rangle$ if and only if the comparator exchanges the order of the two elements input. In more detail, the comparator *first* compares the two elements, storing the resulting bit in the sorting-ancilla; *then* conditioned on the sorting-ancilla being set, it swaps over the two elements. These two steps are each clearly reversible in their own right. The sorting-ancillas are then all retained, to enable ‘unsorting’ later. A reversible comparator for full lexicographic sort on b -bit objects was constructed by Thapliyal et al. [25]. Their algorithm is based on the binary tree and is efficient for our purposes having width $O(b)$ and depth $O(\log b)$.

All of the comparators in any given sorting network should be identical, and when any T elements are input to the sorting network, its overall effect should be to output them in totally sorted order, with certainty. The network can of course be designed completely independently of the comparator, since the details of what makes one element ‘greater’, ‘less’, or ‘equal’ to another is irrelevant from the perspective of which binary comparisons are needed to guarantee sorting. Thus for any value of T , one can ask for the lowest depth sorting network for sorting T elements. In references [2, 22], it is shown that T elements can be sorted in $O(\log T)$ depth of comparators, and that a uniform family of sorting networks achieves this; as needed for our Theorems 2 and 3. Unfortunately the constant for Paterson’s simplified version of the AKS sorting network [22] is around 6,100 and so not practical for any realistic sizes of T . However, the *bitonic sorting network* [18] sorts $T = 2^t$ elements in depth $1 + 2 + \dots + t = t(t+1)/2 = O(\log^2 T)$. The case $T = 8$ is illustrated in Fig. 2.

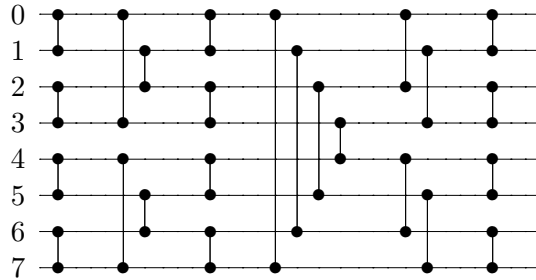


Figure 2: The bitonic sorting network for 8 elements. Each gate in the network represents a comparator between two elements. (The auxiliary registers required to make the overall circuit reversible have been suppressed for clarity.)

Write $s(T)$ for the total number of comparators appearing in a given sorting network for T elements. A sorting operation is written as follows:

$$S_T : |0\rangle \otimes \bigotimes_{k=1}^T |x_k\rangle \mapsto |\sigma\rangle \otimes \bigotimes_{k=1}^T |x_{\sigma(k)}\rangle. \quad (5)$$

Here $\sigma \in \text{Sym}(T)$ denotes a permutation that puts the T elements in order, and the first register (storing σ) actually consists of the $s(T)$ sorting-ancilla bits that get fed into the

comparators. It is clear from this observation that $s(T) \geq \log_2(T!) = \Omega(T \log T)$ for any valid sorting network, since potentially any element of $\text{Sym}(T)$ might be a uniquely correct one. If a sorting network has depth D , then it uses a total of $O(D \cdot T)$ comparators, and so depth must be $D = \Omega(\log T)$ for any valid sorting network.

Because the sorting subroutine is reversible, it makes sense to run it backwards. When that happens, the comparators are encountered in reverse order, and each comparator swaps the order of its inputs according to whether its sorting-ancilla bit is set. That sorting-ancilla bit is then reversibly cleared (regardless of its value) by ‘uncomputing’ the comparison between the two elements.

3.2 The data-moving algorithm

A circuit for data moving, V_N , depends on just two parameters: the number of data registers, N , and the number of bits in each data register, d . That is to say, the same circuit can be used for moving different ‘kinds’ of data, since the circuit treats data items simply as contiguous arrays of d bits per item, regardless of what this data might signify.

The version of a circuit for V_N that we now describe is slightly more complicated than is strictly necessary. But this description has the advantage that it establishes the framework for the parallel look-up circuit, $U_{(N,N)}$ (see §3.3).

It is convenient to break V_N into three basic parts: formatting, sorting, and applying the permutation. The formatting and sorting both need to be reversed after the permutation has been applied. Interestingly, the final formatting is not the exact reverse of the initial formatting, so the algorithm overall isn’t ‘trivially’ a reversible version of a standard classical algorithm. This can all be annotated as follows, reading right to left:

$$V_N = \hat{F} \circ S_{2N}^{-1} \circ P \circ S_{2N} \circ F. \quad (6)$$

We must be careful to count the depth and additional ancilla space required by each of these subroutines (see §3.4).

1) Initial Formatting The subroutine F (described below) can be achieved using SWAP gates and Pauli X gates with no additional ancillas and in depth 1. It is not really a ‘computation’ at all, rather a rearrangement of the input data into a format amenable for describing the sorting that will follow.

Let there be $2N$ ancilla registers that we call *packets*. Each packet contains an *address* ($\lceil \log_2 N \rceil$ bits), a *flag* (1 bit), and a *data* (d bits) register. The initial format moves the

input data out of the input registers and into the packets, ready for sorting:

$$\begin{aligned}
F : |j_1, j_2, \dots, j_N\rangle |x_1, x_2, \dots, x_N\rangle & \bigotimes_{i=1}^N |(0, 0, 0)\rangle_{2i} |(0, 0, 0)\rangle_{2i+1} \\
\mapsto |0, 0, \dots, 0\rangle |0, 0, \dots, 0\rangle & \bigotimes_{i=1}^N |(i, 1, x_i)\rangle_{2i} |(j_i, 0, 0)\rangle_{2i+1},
\end{aligned} \tag{7}$$

where $j_a \neq j_b$ for all $a \neq b$.

One can think of the map F in the following terms. Each processor, i , submits the packet $(i, 1, x_i)$, and if the data it would like to obtain is at index j_i , it also submits the packet $(j_i, 0, 0)$. The flag is used in the sort and unsort steps to record whether the state was originally a storage state (1) or a request for data (0).

2) Sorting Sorting was described in §3.1. In this context, we want to sort the $2N$ packets, using a lexicographical ordering that reads only the address then the flag of each packet. In accordance with Eqn. (5), the subroutine S_{2N} must employ a register of $s(2N)$ sorting-ancilla bits, mapping as

$$\begin{aligned}
S_{2N} : |0\rangle \otimes \bigotimes_{i=1}^N |(i, 1, x_i)\rangle_{2i} |(j_i, 0, 0)\rangle_{2i+1} \\
\mapsto |\sigma\rangle \otimes \bigotimes_{i=1}^N |(i, 0, 0)\rangle_{2i} |(i, 1, x_i)\rangle_{2i+1},
\end{aligned} \tag{8}$$

where $\sigma \in \text{Sym}(2N)$ is the permutation implied by $2i = \sigma(2i + 1)$ and $2i + 1 = \sigma(j_i)$, for i from 1 to N .

The total depth of the circuit for unitary S_{2N} is equal to the depth of the sorting network multiplied by the depth of a single comparator (see §3.4 for a careful analysis).

3) Apply the Permutation After the sort, we are left with a sequence of packets of the form

$$\dots (i-1, 0, 0) (i-1, 1, x_{i-1}) (i, 0, 0) (i, 1, x_i) (i+1, 0, 0) (i+1, 1, x_{i+1}) \dots \tag{9}$$

where the packets are sorted in lexicographical order according to the address and the flag. This ordering makes the permutation step especially simple. Without the need for any auxiliary bits, and in depth 1, SWAP gates can achieve the map

$$P : \bigotimes_{i=1}^N |(i, 0, 0)\rangle_{2i} |(i, 1, x_i)\rangle_{2i+1} \mapsto \bigotimes_{i=1}^N |(i, 0, x_i)\rangle_{2i} |(i, 1, 0)\rangle_{2i+1}. \tag{10}$$

An important property of P is that it does not change the address or the flag used in the sort.

4) Unsorting The sorting network can be run in reverse to return the packets to their original positions. To achieve the unsort, S_{2N}^{-1} acts on the sorting-ancilla register and the packets, mapping

$$\begin{aligned} S_{2N}^{-1} : |\sigma\rangle \otimes \bigotimes_{i=1}^N |(i, 0, x_i)\rangle_{2i} |(i, 1, 0)\rangle_{2i+1} \\ \mapsto |0\rangle \otimes \bigotimes_{i=1}^N |(i, 1, 0)\rangle_{2i} |(j_i, 0, x_{j_i})\rangle_{2i+1}. \end{aligned} \quad (11)$$

Since S_{2N}^{-1} is the same as the sorting network, but with the order of the gates reversed, the cost for the unsort step is the same as for step 2.

5) Final formatting The final step in the algorithm is to write the data back to the original registers and clear the ancilla space. As with the initial formatting, F , the subroutine \hat{F} has depth 1. Acting on the same registers as for its counterpart F , it works as follows:

$$\begin{aligned} \hat{F} : |0, 0, \dots, 0\rangle |0, 0, \dots, 0\rangle \bigotimes_{i=1}^N |(i, 1, 0)\rangle_{2i} |(j_i, 0, x_{j_i})\rangle_{2i+1} \\ \mapsto |j_1, j_2, \dots, j_N\rangle |x_{j_1}, x_{j_2}, \dots, x_{j_N}\rangle \bigotimes_{i=1}^N |(0, 0, 0)\rangle_{2i} |(0, 0, 0)\rangle_{2i+1}. \end{aligned} \quad (12)$$

3.3 The parallel look-up algorithm

We now present the algorithm for accessing memory in parallel, implementing the unitary $U_{(N,N)}$ defined in Eqn. (3). As with V_N , a circuit for $U_{(N,N)}$ depends only on the parameters N and d . We will generalize the algorithm given for V_N to show how $U_{(N,N)}$ may be efficiently implemented.

The overall structure for $U_{(N,N)}$ is the same as for V_N . We first construct packets that include the original data and a flag to be used in the sorting step. After a sorting network is applied, we transform the data. Instead of the permutation used in the data moving algorithm, the transformation to use is composed of *cascading*, B , and *copying*, C . Finally, we reverse the sort and map the data back to the original registers. Accordingly, the parallel look-up algorithm can be written

$$U_{(N,N)} = \hat{F} \circ S_{2N}^{-1} \circ B^{-1} \circ C \circ B \circ S_{2N} \circ F. \quad (13)$$

1) Initial Formatting For the parallel look-up algorithm, we use packets containing four items. Each packet contains an address and flag as before, but now we have two data

registers of d bits, target-data and memory-data, each of d bits. The initial forming stage, F , resembles the same step in §3.2,

$$\begin{aligned}
F &: |j_1, \dots, j_N\rangle |y_1, \dots, y_N\rangle |x_1, \dots, x_N\rangle \bigotimes_{i=1}^N |(0, 0, 0, 0)\rangle_{2i} |(0, 0, 0, 0)\rangle_{2i+1} \\
&\mapsto |0, \dots, 0\rangle |0, \dots, 0\rangle |0, \dots, 0\rangle \bigotimes_{i=1}^N |(i, 1, 0, x_i)\rangle_{2i} |(j_i, 0, y_i, 0)\rangle_{2i+1},
\end{aligned} \tag{14}$$

putting all the data into the packets, where it can be processed by the rest of the algorithm. The initial forming step is achieved in one timestep.

2) Sorting The sorting step is the same as before: we sort lexicographically reading only the address and flag of each packet.

$$\begin{aligned}
S_{2N} &: |0\rangle \otimes \bigotimes_{k=1}^{2N} |(i_k, f_k, y_k, x_k)\rangle_k \\
&\mapsto |\sigma\rangle \otimes \bigotimes_{k=1}^{2N} |(i_{\sigma(k)}, f_{\sigma(k)}, y_{\sigma(k)}, x_{\sigma(k)})\rangle_k.
\end{aligned} \tag{15}$$

Note that packets whose flag is $|1\rangle$ hold data in their memory-data registers, while packets whose flag is $|0\rangle$ hold ‘target data’ in their target-data registers. At the end of the sort, we are left with a sequence of the form

$$\dots (i-1, 0, y, 0)(i-1, 1, 0, x_{i-1})(i, 0, y', 0) \dots (i, 0, y'', 0)(i, 1, 0, x_i)(i+1, 0, y''', 0) \dots \tag{16}$$

3) Cascade The goal of *cascade* is to send a copy of the memory-data registers to the left into the empty memory-data registers of packets that have their flags set to $|0\rangle$. Since there is no way of knowing in advance how far the data will need to propagate, we need a method that works in all cases. For example, every processor could request data from a single processor, say $j_0 = j_1 = \dots = j_{N-1} = 1$. This can be achieved by dividing the cascade up into n smaller *phases*, where $n = \lceil \log_2(2N) \rceil$. Accordingly we write

$$B = B_{n-1} \circ B_{n-2} \circ \dots \circ B_1 \circ B_0, \tag{17}$$

where each B_k phase acts on pairs of packets at relative index separation 2^k from one another, more or less in parallel.

To achieve this transformation, each packet will need a fresh ancilla *aux-phase* register of $\lceil \log_2 n \rceil$ bits. This will be used to store a number indicating the phase in which that packet acquires a copy of the data being cascaded to it. This record makes it much easier

to implement the whole cascade reversibly, and these aux-phases are to be retained until the cascade is later reversed. Each packet will also need an *aux-action* bit that persists only throughout a single phase (recycled from one phase to the next), being set if that packet receives data during that phase.

Each phase involves many pairs of packets. Phase B_k involves pairs with indices l and $l + 2^k$, for all l for which $l \geq 0$ and $l + 2^k < 2N$. If $\lfloor 2^{-k}l \rfloor$ is even (*resp.* odd), then the pair $(l, l + 2^k)$ is said to be of *even* (*resp.* *odd*) *parity*. The idea is that data can legitimately cascade from the packet at $l + 2^k$ to the one at l if they have the same address and if the rightmost one presently has data but the leftmost one doesn't. It is enough to check that the leftmost one has $|0\rangle$ for its flag and $|0\rangle$ for its aux-phase, and that the rightmost one has $|1\rangle$ for its flag or something non-zero for its aux-phase. If that overall condition is met, flip the aux-action bit of the leftmost packet, because it will be receiving data this phase. It is necessary to set the aux-actions for all the even parity pairs first, then for all the odd parity pairs, because otherwise the same bits will be being read by different gates at the same time, and that violates the rules of the standard circuit model. Note also that the simple act of computing the aux-action bit may itself require a little extra ancilla scratch space.

When *all* the aux-action bits have been correctly set for the present phase, cascade data leftwards, locally conditioned on those aux-action bits. For example, during phase B_k when examining packets l and $l + 2^k$, if l has had its aux-action set, then we need to implement

$$\begin{aligned} |(i, 0, y', 0)\rangle_l |0\rangle_{\text{phase}(l)} \otimes |(i, f, y, x)\rangle_{l+2^k} |p\rangle_{\text{phase}(l+2^k)} \\ \mapsto |(i, 0, y', x)\rangle_l |k+1\rangle_{\text{phase}(l)} \otimes |(i, f, y, x)\rangle_{l+2^k} |p\rangle_{\text{phase}(l+2^k)}. \end{aligned} \quad (18)$$

(The aux-action for l got set because one of f and p was non-zero.) This should be done for the even parity pairs first (say), then the odd parity ones.

Finally, the aux-action bits need to be reset. This resetting is a 'local' operation, because during phase B_k , each packet need flip its aux-action if and only if its aux-phase is $|k+1\rangle$, indicating that it was active this phase. Note that the condition for resetting an aux-action bit is completely different from the condition for setting it in the first place.

The total effect of B will be to load up the aux-phase ancillas and to replace every instance of $|(j_i, 0, y_i, 0)\rangle$ with $|(j_i, 0, y_i, x_{j_i})\rangle$, while preserving every instance of $|(i, 1, 0, x_i)\rangle$.

4) Copying C is a simple depth 1 local operation. Every packet simply xors the contents of its memory-data into its target-data. This has the effect of mapping every $|(j_i, 0, y_i, x_{j_i})\rangle$ to $|(j_i, 0, y_i \oplus x_{j_i}, x_{j_i})\rangle$, while every $|(i, 1, 0, x_i)\rangle$ gets mapped to $|(i, 1, x_i, x_i)\rangle$.

5) Reversing the Cascade This map reverses the effect of the cascade, cleaning up all the aux-phase ancillas, making the packets ready for unsorting.

The total effect of $(B^{-1} \circ C \circ B)$ is to replace every instance of $|j_i, 0, y_i, 0\rangle$ with $|j_i, 0, y_i \oplus x_{j_i}, 0\rangle$, while replacing $|i, 1, 0, x_i\rangle$ with $|i, 1, x_i, x_i\rangle$ for all i . This action does not change the ordering of the packets that only depends on the address and flag of each packet. Therefore the action is compatible with the sorting stages, as required.

6) Unsort The unitary S_{2N}^{-1} unsorts just as before.

7) Final Formatting The final step is to apply a formatting map, \widehat{F} , which works as follows,

$$\begin{aligned}
& |0, \dots, 0\rangle |0, \dots, 0\rangle |0, \dots, 0\rangle \bigotimes_{i=1}^N |(i, 1, x_i, x_i)\rangle_{2i} |(j_i, 0, y_i \oplus x_{j_i}, 0)\rangle_{2i+1} \\
& \mapsto |j_1, \dots, j_N\rangle |y_1 \oplus x_{j_1}, \dots, y_N \oplus x_{j_N}\rangle |x_1, \dots, x_N\rangle \bigotimes_{i=1}^N |(0, 0, 0, 0)\rangle_{2i} |(0, 0, 0, 0)\rangle_{2i+1}.
\end{aligned} \tag{19}$$

Note that this is a depth 2 map rather than a depth 1 map because each x_i appears in two places, and these need to ‘relocalise’ as well as ‘move’.

3.4 Proof of Theorems 2 and 3

Here we count up the total depth and width of circuits for V_N and for $U_{(N,N)}$, thereby proving Theorems 2 and 3.

The total depth of the formatting subroutines is $O(1)$.

The sorting subroutines require comparators that make a lexicographic comparison on $\lceil \log_2 N \rceil + 1$ bits, and lexicographic comparison is basically the first part of *arithmetic subtraction* (regard the bit-patterns as integers, subtract one from the other reversibly, read the sign bit of the output). This can be achieved efficiently in depth $O(\log \log N)$ [25]. The other thing a comparator does is to swap elements controlled on a sorting-bit. The sizes of our elements are $O(\log N + d)$, and so the sorting bit needs to be fanned out (using a binary tree) to ‘copy’ it across $O(\log N + d)$ ancillas, before a depth 1 swap can take place. Therefore the swapping stage of a comparator takes depth $O(\log(\log N + d))$, and so the total comparator has depth $O(\log \log N + \log(\log N + d)) = O(\log(d \log N))$. Since there are $O(\log N)$ comparators in the AKS sorting network, the total depth of the sorting stage is $O(\log N \cdot \log(d \log N))$.

The cascade subroutine has $n = O(\log N)$ phases, and as with the comparators used in sorting, each phase of cascade involves arithmetic (in fact equality testing) on objects of size $O(n)$, plus controlled copying of objects of size $O(d)$. Therefore a phase has

depth $O(\log n + \log d) = O(\log(d \log N))$ and the total depth of the cascade part is $O(\log N \cdot \log(d \log N))$.

The inner subroutine (‘copying’) of the algorithm for $U_{(N,N)}$ has depth $O(1)$ and requires no ancillas.

Therefore the total depth of our algorithm for $U_{(N,N)}$ is $O(\log N \cdot \log(d \log N))$, and for V_N it is of the same order.

Counting bits, we see that besides the $O(N(\log(N) + d))$ input/output bits, we require $O(N(\log(N) + d))$ more bits for the packets, $O(N \log N)$ sorting-ancilla bits, as well as $O(N(\log(N) + d))$ ancilla bits for temporary use while rendering the (lexicographic) comparators. Furthermore, for *cascade* each packet needs an aux-phase register and an aux-action bit (total $O(N \log(N))$ bits), as well as $O((\log(N) + d))$ scratch space (*e.g.* to compute/reset the aux-action bit) that can be recycled between phases. Hence, the total circuit width is $O(N(\log(N) + d))$, which is linear in the width of the logical unitary in question, and therefore asymptotically optimal. \square

4 Distributed Quantum Computing

The circuit model is *universal* for quantum computing, with the gate set consisting of all single-qubit unitaries and the 2-qubit CNOT gate, for example.

The circuit model allows any pair of qubits to be connected by gates and so allows arbitrarily long-range interactions. This model is very far from any likely implementation of a quantum computer. We imagine that a small number of long-range interactions could be possible, but most gates will need to be local.

It is well known that if the qubits are laid out in a line and each could only connect with its nearest neighbours (one either side), then the resulting model of computation would still be universal, because it could emulate any circuit of the more general kind. The emulation proceeds in a straightforward fashion using SWAP gates to bring qubits next to each other so that nearest-neighbor gates can be applied. The price to pay in this emulation is (in general) an overhead factor of $O(W)$ in the depth of the algorithm, where W counts the total width (number of qubits) of the circuit being emulated.

More generally, we could envisage having memory larger than a single qubit at each ‘site’, with connectivity more generous than simply being connected each to two neighbours in a line. Then the *overhead depth factor* (or *emulation factor*) of universal emulation could generally be reduced to something smaller than $O(W)$. Our design of §3.2 is useful in this context, as explained next.

4.1 Efficient Distributed Quantum Computing

Considering a general quantum circuit of width W , let there be N quantum processors, each with its own local memory of Q qubits. Suppose that $Q \cdot N \geq W$ and suppose the processors are interconnected as the N nodes of a graph \mathcal{G} . We say that a circuit *respects graph locality* if every two-qubit gate in the circuit *either* has those two qubits lying in the same processor's local memory *or else* has them lying in neighbouring processors' memories (neighbouring with respect to \mathcal{G}). In addition, we require that each gate is assigned to a processor that holds at least one of its qubits, and each processor can only perform one gate per timeslice. Together, these restrictions on the ordinary circuit model define the distributed quantum computing model formally.

What then is the best overhead depth factor for arbitrary circuit emulation in the worst case? Starting with an arbitrary circuit of W qubits (one perhaps not respecting the locality of \mathcal{G}), we wish to emulate it using a circuit that respects graph locality. We want the overhead depth factor of this emulation to be as small as possible. That is, we wish to minimise the function

$$F(\mathcal{G}, Q, W) := \max_C \frac{\text{Depth}(C')}{\text{Depth}(C)} \quad (20)$$

where C' is a circuit for emulating C subject to the constraints imposed by \mathcal{G} and Q . Maximisation is over all circuits C of width W so $F(\mathcal{G}, Q, W)$ is the worst case cost of emulating arbitrary circuits. Normally we are concerned with the case $W = |\mathcal{G}| = N$, so that an emulation always has one processor per qubit being emulated, and where Q is large enough to hold ancilla for basic computations.

The emulation factor, $F(\mathcal{G}, Q, W)$, must be at least the diameter of \mathcal{G} , and that in turn depends on the order, N , and valency, $\text{val}(\mathcal{G})$, of the graph in question. The following Theorem demonstrates a nice trade-off reducing the overhead depth factor significantly, while keeping both Q and $\text{val}(\mathcal{G})$ 'reasonably' small.

Theorem 4. *For a distributed quantum computer with $|\mathcal{G}| := N$, we can find a graph \mathcal{G} for which $\text{val}(\mathcal{G}) = O(\log N)$, and take $Q = O(\log N)$, and yet have overhead depth factor $F(\mathcal{G}, Q, N) = \tilde{O}(\log^2 N)$ for emulating arbitrary quantum circuits of width N .*

Proof : The proof of the Theorem uses our algorithm for V_N (cf. §§2.3, 3.2). The proof is clearest when we take $d = 1$, but other settings are possible. In overview, let each processor be big enough to hold two packets and some ancillas, *i.e.* $Q = O(d + \log N)$. First we show how V_N with data size $d = 1$ can be efficiently implemented in this processor model (*i.e.* as a circuit respecting graph locality), where the data bits are distributed one per processor. This is more or less a direct embedding of our algorithm for V_N into the processor model, and serves to define the graph \mathcal{G} of the Theorem. Then we show how any general circuit of width N can be efficiently fit into the processor model, emulated in terms of intra-processor gates and renditions of V_N only.

Implementing V_N To implement V_N across N processors, start by putting one data bit per processor. We need to implement subroutines **Format**, **Sort**, and **Permutation** (cf. Eqn. (6)). The first and last of these are entirely local operations, so their circuits are already admissible to the processor model.

It only remains to check that the circuit for the sorting subroutine is admissible within the processor model, and that of course depends on the graph \mathcal{G} . The gates of this sorting subroutine all belong to comparators of the sorting network. Each comparator can be admitted within the processor model provided that the topology of \mathcal{G} is inherited from the topology of the sorting network. The sorting network itself has depth $O(\log N)$ in the optimal case, and so there exists a graph \mathcal{G} with valency $O(\log N)$ with regard to which our algorithm for V_N embeds unaltered.

Since there can only be one gate per processor per timeslice in this model, the depth of V_N goes up from $O(\log N \cdot \log(d \log N))$ to $\tilde{O}(\log N \cdot (d + \log N))$. This latter figure may be written $\tilde{O}(\log^2 N)$ when $d = 1$.

Emulating general circuits via V_N Given a general circuit of width N , in each timeslice there will be at most N gates. These gates should be ‘assigned’ one per processor when the timeslice is emulated. When a processor comes to emulate a gate assigned to it, it will need access to the one or two qubits of that gate. The emulation of a timeslice therefore requires two calls to the subroutine V_N : without loss of generality, we can assume that the first qubit of a gate already resides at the processor to which that gate has been assigned; the first call to V_N brings the second qubit of each gate to its processor; the processor implements the gate locally on the two qubits; the second call to V_N restores the second qubit of each gate to its original home. Null ancilla qubits can be included within each V_N operation in order to make it a permutation.

Every timeslice of the circuit being emulated now additionally requires two calls to V_N , plus appropriate $O(\log N)$ -sized circuitry (per processor) to write and erase the indices j_i used within V_N . Overall this costs an overhead depth factor of $\tilde{O}(\log^2 N)$. This completes the proof of Theorem 4. \square

4.2 Implementing quantum circuits on any architecture

In §3.1, we pointed out that for any (positive integer) T there must be a lowest-depth sorting network for sorting T elements deterministically. More generally, one can say that for any connected graph \mathcal{G} of T vertices, there must exist a lowest-depth sorting network for sorting T elements, all of whose comparators lie along edges of \mathcal{G} . For example, when $T = 4$ and \mathcal{G} is a 4-cycle, then the bitonic sorting network fits the graph, and, having six comparators in depth 3, is optimal. But for $T = 4$ and \mathcal{G} a graph of three edges in a line, the bitonic sort is not possible, and a sort involving six comparators in depth 4 turns out to be optimal for this graph.

In Theorem 4, we constrained the valency ($val(\mathcal{G})$) and order (N) of \mathcal{G} , but otherwise permitted any connected graph, and for the proof of the Theorem we used the graph implied by the best known sorting network for that N . This concept can be generalised to any given connected graph \mathcal{G} on N vertices. The problem of identifying the best sorting algorithm for N items that is compatible with \mathcal{G} is essentially the same as identifying the most efficient circuit for implementing V_N in the distributed quantum computing model. Hence, the overhead depth factor associated with the distributed quantum computing model employing graph \mathcal{G} is related to the cost of the optimal sorting algorithm over \mathcal{G} . Scheduling tasks on topologically constrained (quantum) computing platform may be regarded as (more or less) equivalent to the problem of designing sorting algorithms (including sorting networks) commensurate with those topologies. We put these ideas on a firmer footing in the following Theorem.

Theorem 5. *Let \mathcal{G} be any connected graph on N nodes, and consider the distributed quantum computing model with graph \mathcal{G} and $Q = O(\log N)$ qubits per processor. Let $D_{\mathcal{G}}$ denote the depth of the best algorithm for sorting $2N$ arbitrary bit strings of length $\lceil \log_2 N \rceil + 2$ over \mathcal{G} (with Q qubits per node). Then the minimum depth overhead, $F_{min}(\mathcal{G}, Q, W)$, for emulating arbitrary circuits is bounded by*

$$F_{min}(\mathcal{G}, Q, N) \leq O(D_{\mathcal{G}}) \quad (21)$$

and

$$O\left(\frac{D_{\mathcal{G}}}{\log N \log \log N}\right) \leq F_{min}(\mathcal{G}, Q, O(N \log N)). \quad (22)$$

Proof : To prove the upper bound (21) we give an explicit construction analogous to the one used in Theorem 4. It emulates a circuit using an efficient sorting algorithm whose depth is given by $D_{\mathcal{G}}$. For any circuit C of width $W = N$ qubits, we construct its emulator circuit, C' , by assigning each qubit and gate of C to a processor of C' and then use an implementation of V_N (with $d = 1$) to move qubits around so that gates are always local. Recall from §3.2 that our description of V_N required sorting $2N$ packets, each of size $\lceil \log_2 N \rceil + 1 + d$ bits.

The rest of the proof for the upper bound then mirrors the proof for Theorem 4. As before, the forming and copying steps take time $O(\log N)$, the only difference here is that the sort is achieved in depth $D_{\mathcal{G}}$. Hence the cost of this emulation algorithm is $F(\mathcal{G}, Q, N) = O(\log N + D_{\mathcal{G}}) = O(D_{\mathcal{G}})$ since $O(D_{\mathcal{G}}) \geq O(\log N)$.

For the lower bound (22), consider the AKS sorting network for sorting $2N$ packets of bit-length $\lceil \log_2 N \rceil + 2$. Let C be the (unconstrained) circuit for achieving this, so the width of C is $W = O(N \log N)$ and its depth is $O(\log N \log \log N)$ (cf. §3.4). The cost of emulating C is bounded by $D_{\mathcal{G}}$ since the emulation is a sorting algorithm on the (\mathcal{G}, Q) -distributed computer and we defined $D_{\mathcal{G}}$ to be the depth of the best sorting algorithm. Hence the cost of any emulation is lower bounded by $O(D_{\mathcal{G}} / \log N \log \log N)$, as required. \square

5 Revisiting Popular Algorithms

Grover's quantum search algorithm [16] and the generalization to amplitude amplification [7, 8] have the great advantage of being widely applicable. Problems such as element distinctness [10, 4] collision finding [9], triangle finding [5, 20], 3-SAT [3] and NP-hard tree search problems [11] all have faster quantum algorithms because they are able to make use of amplitude amplification. In this Section, we revisit Grover's quantum search algorithm and the Element Distinctness problem, resolving a challenge posed by Grover and Rudolph [17].

5.1 Quantum search of an unstructured database

Grover's fast quantum search algorithm [16] is usually presented as solving an oracle problem. We are presented with a function $f : \{1, \dots, N\} \rightarrow \{0, 1\}$ and wish to find solutions s_j such that $f(s_j) = 1$. We also have an oracle with the ability to recognise solutions s_j in the form of a unitary

$$U_f : |y\rangle|b\rangle \mapsto |y\rangle|b \oplus f(y)\rangle. \quad (23)$$

Setting the target register, $|b\rangle$, of U_f to the state $(|0\rangle - |1\rangle)/\sqrt{2}$ encodes the value of $f(y)$ into a phase shift

$$U_f : |y\rangle \mapsto (-1)^{f(y)}|y\rangle, \quad (24)$$

where we have suppressed the target register since it remains unchanged. Grover's algorithm then makes $O(\sqrt{N/M})$ calls to U_f to find one of the M possible solutions, s_j , uniformly at random. To find more solutions we simply repeat the algorithm, and by a coupon-collector argument [13], we find r solutions in overall circuit depth $O(r \log r \sqrt{N/M})$. Throughout this Section, we set the width and depth of a function evaluation to be $O(1)$, since the cost is simply an overall multiplicative factor.

Grover's algorithm can be applied to search an unstructured database. We construct the oracle by using the single memory look-up unitary, $U_{(1,N)}$, together with a simple function that compares an input to the database entry (see Chap. 6.5 in [21]). More generally, suppose we wish to find solutions to a function whose inputs cannot be simply expressed as the numbers from 1 to N , but rather are taken from a database of elements $X = \{x_j : j = 1, \dots, N\}$, where each x_j is a bit string of length d . That is, we have a function $\alpha : X \rightarrow \{0, 1\}$, and are searching for solutions $x_j \in X$ such that $\alpha(x_j) = 1$. Once the database item x_j has been loaded into the computational memory, the function α is computed using the unitary

$$U_\alpha : |j\rangle|x_j\rangle|b\rangle \mapsto |j\rangle|x_j\rangle|b \oplus \alpha(x_j)\rangle. \quad (25)$$

We consider the case where the database is held in a quantum state $|x_1, \dots, x_N\rangle$, but it could also be a classical database whose indices can be accessed in superposition [14, 15].

An oracle is constructed by first looking-up a database entry and then testing if this entry is a solution to the function α . The initial state of the computer is

$$|j\rangle|0\rangle|b\rangle|x_1, \dots, x_N\rangle \quad (26)$$

where the state $|j\rangle$ is the index of the database item we will look-up and $|0\rangle$ and $|b\rangle$ are auxiliary states used to load the memory and store the result respectively.

First we apply the single memory look-up unitary, $U_{(1,N)}$, (see Definition. 1) so that the computer is in the state

$$|j\rangle|x_j\rangle|b\rangle|x_1, \dots, x_N\rangle, \quad (27)$$

then calling the function α , using the corresponding unitary U_α , maps the state to

$$|j\rangle|x_j\rangle|b \oplus \alpha(x_j)\rangle|x_1, \dots, x_N\rangle. \quad (28)$$

Finally we restore the auxiliary state used to load the database item by applying $U_{(1,N)}^\dagger = U_{(1,N)}$. The final state of the computation is therefore

$$|j\rangle|0\rangle|b \oplus \alpha(x_j)\rangle|x_1, \dots, x_N\rangle. \quad (29)$$

Hence the unitary

$$\mathcal{O}_\alpha = U_{(1,N)} \circ U_\alpha \circ U_{(1,N)} \quad (30)$$

can be used as an oracle for Grover's algorithm. Setting the target register to $(|0\rangle - |1\rangle)/\sqrt{2}$ encodes the value of $\alpha(x_j)$ into a phase shift

$$\mathcal{O}_\alpha : |j\rangle \mapsto (-1)^{\alpha(x_j)}|j\rangle. \quad (31)$$

The quantum circuit implementing Grover's algorithm with \mathcal{O}_α as an oracle requires circuit width $O(N \log N)$ and depth $O(\sqrt{N/M} \log N)$ to find one solution.

Now that we have an efficient algorithm for performing parallel memory look-ups, we consider the effect of using the unitary $U_{(N,N)}$ together with (up to) N functions as an oracle for Grover's algorithm.

Suppose we have (up to) N functions that can take database elements as inputs, $\alpha_i : X \rightarrow \{0, 1\}$, for $i = 1, \dots, N$. Just as with a standard Grover search, each α_i can be any function provided it is computable in time polynomial in $\log N$ (and d). We wish to find solutions $x_{j_i} \in X$ such that $\alpha_i(x_{j_i}) = 1$, for all $i = 1, \dots, N$. The following theorem provides an algorithm that finds (up to) N solutions, one for each function, using the same size quantum circuit (up to log factors) as would be required to find only one solution using the unitary $U_{(1,N)}$ as an oracle.

Theorem 6 (Multi-Grover search algorithm). *Using the notation defined above, there is a quantum algorithm that for each $i = 1, \dots, N$ either returns j_i such that $\alpha_i(s_{j_i}) = 1$, or, if there is no such solution, returns 'no solution'. The algorithm can be implemented using a quantum circuit with width $\tilde{O}(N)$ and depth $\tilde{O}(\sqrt{N})$.*

Proof : We proceed as with the single Grover search of an unstructured database. The only subtlety being that we need to organise the Hilbert space in the correct way. We want to perform N Grover searches over the database so we need N indices $|j_1 \dots j_N\rangle$, N memory place holders $|0 \dots 0\rangle$ and N target registers $|b_1 \dots b_N\rangle$.

It is useful to split the circuit in to N ‘processors’ since we will think of each one as performing a Grover search over the database. We rearrange the Hilbert space across N processors as

$$|j_1, \dots, j_N\rangle |0, \dots, 0\rangle |b_1, \dots, b_N\rangle |x_1, \dots, x_N\rangle \cong \bigotimes_{i=1}^N |j_i, 0, b_i, x_i\rangle. \quad (32)$$

Applying the parallel look up algorithm, maps the initial state of the computer to

$$\bigotimes_{i=1}^N |j_i, x_{j_i}, b_i, x_i\rangle. \quad (33)$$

We define a circuit for implementing the unitaries U_{α_i} in parallel as

$$U_{\alpha} = \bigotimes_{i=1}^N U_{\alpha_i}, \quad (34)$$

which acts on all of the target register simultaneously sending the state to

$$\bigotimes_{i=1}^N |j_i, x_{j_i}, b_i \oplus \alpha_i(x_{j_i}), x_i\rangle. \quad (35)$$

Finally, we clear the auxiliary register used to load the memory item using the unitary $U_{(N,N)}$, so that the final state of the computer is

$$\bigotimes_{i=1}^N |j_i, 0, b_i \oplus \alpha_i(x_{j_i}), x_i\rangle. \quad (36)$$

Setting the target registers $(|0\rangle - |1\rangle)/\sqrt{2}$ produces the required oracle using a circuit with width and depth $O(N \log N)$ and $O(\log N \log \log N)$, respectively (c.f. Theorem 2).

Grover’s algorithm then calls the unitary

$$U_{(N,N)} \circ U_{\alpha} \circ U_{(N,N)} \quad (37)$$

\sqrt{N} times. The resulting algorithm finds one solution for each function α_i using a circuit with width $O(N \log N)$ depth $O(\sqrt{N} \log N \log \log N)$. \square

If X were highly structured (such as being the numbers 1 to N), it would be straightforward to perform $\tilde{O}(N)$ Grover searches in parallel, using a circuit of width $O(N)$, since we would not need to store X explicitly. Now, making use of the efficient parallel memory look-up algorithm to access X , we are able to interlace the steps in the Grover algorithm with database look-ups. The end result is that we can indeed perform $\tilde{O}(N)$ Grover searches in parallel regardless of the structuring of X . In the next subsection we examine the effect of Theorem 6 on existing memory intensive quantum algorithms.

5.2 Element Distinctness

In this Section, we present a quantum algorithm for the element distinctness problem: given a function $f : \{0, 1\}^n \rightarrow \{0, 1\}^n$, determine whether there exists distinct $i, j \in \{0, 1\}^n$ with $f(i) = f(j)$.

The size of the problem is parametrised by $N = 2^n$, and let S denote the available memory, measured in n -bit words. Suppose that f can be evaluated in time and space $O(1)$. Previous quantum algorithms for element distinctness require time T satisfying

$$ST^2 = \tilde{O}(N^2). \quad (38)$$

Buhrman et al [10] achieve this for S up to $N^{1/2}$; Ambainis [4] extends this to S up to $N^{2/3}$.

The Buhrman et al and Ambainis algorithms are for single processor machines. Grover and Rudolph [17] have pointed out that the notion of a single processor machine with large memory makes little sense in the quantum world. They argue that requiring space S is no better than using S processors, and show that for any S , the trade-off in Eqn. (38) can be achieved by simply having each processor apply Grover's search algorithm to a search space of size $O(N^2/S)$. Grover and Rudolph pose beating this trade-off as a challenge.

Our algorithm answers this challenge: we achieve the trade-off

$$ST = \tilde{O}(N) \quad (39)$$

for essentially all S up to N . It can be thought of in the processor model (like the Grover-Rudolph algorithm) with k processors each using space S/k and total depth $\tilde{O}(N/S)$, for some k . It is a variant of the Buhrman et al algorithm [10], requiring processors to access a *shared* memory of size $\tilde{O}(k)$. Equivalently we can describe it in the circuit model using calls to $U_{(k,k)}$ with total width S and total depth $\tilde{O}(N/S)$. (Theorem 4 guarantees that conversion factors between these two models are polylogarithmic.)

We begin by constructing a database of function evaluations $X = \{f(j) : j = 1 \dots N\}$ which takes time N/k since we have k processors. Now consider the following algorithm that checks if the first k items are marked.

- Sort the numbers $f(1), \dots, f(k)$ and check if any of them are equal. This can be achieved using the reversible AKS sorting network and so takes time $T = \tilde{O}(\log k)$.
- Using the sorted list, L , we construct a function, $g : L \times X \rightarrow \{0, 1\}$, defined by

$$g(f(i), f(j)) = \begin{cases} 1 & \text{if } f(i) = f(j) \\ 0 & \text{otherwise.} \end{cases} \quad (40)$$

Since L is sorted, g can be computed in time $T = O(1)$.

- Using the multi-Grover algorithm given in Theorem 6, we can search the remaining $N - k$ database elements in time $T = O\left(\sqrt{\frac{N-k}{k}}\right)$.

This algorithm checks if any of the first k indices result in a match, $f(i) = f(j)$ for $i = 1 \dots k$ and $j = 1 \dots N$. It succeeds with probability N/k and can be repeated for any block of k indices. We can use the algorithm as a Grover oracle so that calling this oracle $\sqrt{N/k}$ times solves the element distinctness problem. The overall time taken is

$$T = O\left(\frac{N}{k} + \sqrt{\frac{N}{k}} \sqrt{\frac{N-k}{k}}\right) = O\left(\frac{N}{k}\right) \quad (41)$$

The above discussion proves the following theorem.

Theorem 7. *For any $k \leq N$, the distinctness of N n -bit strings, can be decided by a quantum circuit with width $O(k \cdot \log N)$ and depth $\tilde{O}((N/k) \log k)$, resulting in the trade-off*

$$ST = \tilde{O}(N).$$

5.3 Collision problem

Our results also apply to the collision problem: in which $f : [N] \mapsto [N]$ is an efficiently-computable function with the promise of being either 1-1 or 2-1, for which an $ST = O(N^{2/3})$ query algorithm is given in [9]. This problem may be solved with

$$ST = \tilde{O}(\sqrt{N})$$

either by selecting $O(\sqrt{N})$ random elements and solving element distinctness, or by simply using the algorithm of [9] directly, augmented by using S processors with shared memory together with our look-up algorithm. So we perform S Grover searches in parallel on spaces of size N/S^2 instead of a single search on a space of size N/S .

6 Conclusion and discussion

We have presented a new algorithm for accessing quantum memory in parallel. The algorithm is extremely efficient; it has an overhead that is scarcely larger than *any* algorithm capable of accessing even a single entry from memory.

The first application of this algorithm is to distributed quantum computing that is constrained to respect the locality of a graph. A variant of the parallel look-up algorithm, which we call the data-moving algorithm, provides an efficient way of mapping circuits respecting only the complete graph to circuits respecting this limited graph. In Theorem 4, we presented a particularly nice situation where the properties of the limited graph are balanced with the cost of emulating the circuit model. Each of the N processors contains $O(\log N)$ qubits and has $O(\log N)$ connections to other processors and yet arbitrary quantum circuits can be emulated with an overhead of $O(\log^2 N)$.

One can think of our data-moving algorithm and Theorem 5 as a proposal for an efficient distributed quantum computer. An architecture based on the bitonic sorting network (we use the bitonic network since the constants for the asymptotically optimal AKS network are too large), would be able to efficiently simulate any algorithm presented in the circuit model.

The idea of using sorting networks as a tool for constructing efficient quantum communication protocols opens up many interesting questions and possibilities for future research. For example, we note that the parallel look-up map (Definition 2) is an example of ‘pull’ map: each ‘instruction’ j_i has a *location* and a *value*; the location describes the *destination* of a data item being transferred (*i.e.* where it’s being pulled to), while the value describes the *source* of that data item (*i.e.* where it’s being pulled from). Our algorithm can be extended to perform analogous ‘push’ maps, where the roles of *destination* and *source* are exchanged (see Appendix A for more details). More generally, it provides a framework for efficient communication in distributed quantum computing.

We have demonstrated that the parallel look-up algorithm can be used to optimize existing quantum algorithms. We provided an extension of Grover’s algorithm that efficiently searches over a physical database for multiple solutions, and answered an open problem posed by Grover and Rudolph by demonstrating an improved space-time trade-off for the Element Distinctness problem. It seems likely that this framework for efficient communication in parallel quantum computing will be a useful subroutine in other memory-intensive quantum algorithms too, such as triangle finding, or more generally for frameworks such as learning graphs.

Acknowledgments

The authors would like to thank the Heilbronn Institute for Mathematical Research for hosting the discussions that led to this research.

AWH was funded by NSF grants 0916400, 0829937, 0803478, DARPA QuEST contract FA9550-09-1-0044 and IARPA via DoI NBC contract D11PC20167. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of IARPA, DoI/NBC, or the U.S. Government.

References

- [1] D. Aharonov, M. Ben-Or, R. Impagliazzo, and N. Nisan, *Limitations of noisy reversible computation*, arXiv:quant-ph/9611028 (1996)
- [2] M. Ajtai, J. Komlos, and E. Szemerédi, *An $O(n \log n)$ sorting network*, Proc. 15th annual ACM symposium on Theory of computing, 1 (1983)
- [3] A. Ambainis, *Quantum Search Algorithms*, SIGACT News, **35** 22 (2004)
- [4] A. Ambainis, *Quantum walk algorithm for element distinctness*, SIAM Journal on Computing **37**, 210-239 (2007)
- [5] A. Belovs, *Span Programs for Functions with Constant-Sized 1-certificates* STOC '12 Proceedings of the 44th symposium on Theory of Computing, 77 (2012)
- [6] A. Blais, R.-S. Huang, A. Wallraff, S. Girvin, and R. Schoelkopf, *Cavity quantum electrodynamics for superconducting electrical circuits: An architecture for quantum computation*, Phys. Rev. A **69**, 062320 (2004)
- [7] G. Brassard and P. Hoyer, *An exact quantum mechanical polynomial time algorithm for Simon's problem*, Proc ISTCS '97, 12 (1997)
- [8] G. Brassard, P. Hoyer, M. Mosca, and A. Tapp, *Quantum amplitude amplification and estimation*, Quantum Computation and Quantum Information Science, AMS Contemporary Math series (2000)
- [9] G. Brassard, P. Hoyer, and A. Tapp, *Cryptology Column - Quantum algorithms for the collision problem*, ACM SIGACT News **28**, 14 (1997)
- [10] H. Buhrman, C. Durr, M. Heiligman, P. Hoyer, F. Magniez, M. Santha, and R. de Wolf, *Quantum Algorithms for Element Distinctness*, SIAM J. Comp. **34**, 1324 (2005), arXiv:quant-ph/0007016
- [11] N. Cerf, L. Grover, and C. Williams, *Nested quantum search and NP-hard problems*, Applicable Algebra in Engineering, Communication and Computing, **10**, 311 (2000)

- [12] J. Cirac, P. Zoller, H. Kimble, and H. Mabuchi, *Quantum State Transfer and Entanglement Distribution among Distant Nodes in a Quantum Network*, Phys. Rev. Lett. **78**, 3221 (1997)
- [13] W. Feller, *An Introduction to Probability Theory and its Applications, Vol 1*, John Wiley and Sons, 1950.
- [14] V. Giovannetti, S. Lloyd, and L. Maccone, *Quantum random access memory*, Phys. Rev. Lett. **100**, 160501 (2008).
- [15] V. Giovannetti, S. Lloyd, and L. Maccone, *Architectures for a quantum random access memory*, Phys. Rev. A **78**, 052310 (2008)
- [16] L. Grover, *A fast quantum mechanical algorithm for database search*, Proc. 28th Annual ACM STOC, 212 (1996)
- [17] L. Grover and T. Rudolph, *How significant are the known collision and element distinctness quantum algorithms?*, Quantum Information & Computation **4**, 201 (2004) arXiv:quant-ph/0309123
- [18] D. Knuth, *The art of computer programming: volume 3*, Addison-Wesley, 1998.
- [19] F.T. Leighton, *Introduction to parallel algorithms and architectures*, Morgan Kaufman Publishers, San Mateo, CA. (1992)
- [20] F. Magniez, M. Santha and M. Szegedy, *Quantum algorithms for the triangle finding problem*, SIAM Journal of Computing **37**, 413 (2007)
- [21] M. Nielsen and I. Chuang, *Quantum computation and Quantum Information*, Cambridge University Press 2000.
- [22] M. Paterson, *Improved Sorting Networks with $O(\log N)$ Depth*, Algorithmica **5**, 75 (1990)
- [23] H. Ramesh and V. Vinay, *String matching in $\tilde{O}(\sqrt{n}+\sqrt{m})$ quantum time*, J. Discrete Algorithms **1**, 103 (2003)
- [24] S. Ritter, C. Nölleke, C. Hahn, A. Reiserer, A. Neuzner, M. Uphoff, M. Mücke, E. Figueroa, J. Bochmann, and G. Rempe, *An elementary quantum network of single atoms in optical cavities*, Nature **484**, 195 (2012)
- [25] H. Thapliyal, N. Ranganathan, and R. Ferreira, *Design of a Comparator Tree Based on Reversible Logic*, Proc. 10th IEEE Conference on Nanotechnology (IEEE-NANO), 1113 (2010)
- [26] L. G. Valiant. *Optimally Universal Parallel Computers* Proc. R. Soc. A, Vol. 326, No. 1591, Solving Scientific Problems on Multiprocessors (Sep. 26, 1988), pp. 373-376
- [27] L. G. Valiant. *A bridging model for parallel computation*, Commun. ACM **33**, 8 (August 1990), 103-111.

- [28] A. Wallraff, D. Schuster, A. Blais, L. Frunzio, R.-S. Huang, J. Majer, S. Kumar, S. Girvin, and R. Schoelkopf, *Strong coupling of a single photon to a superconducting qubit using circuit quantum electrodynamics*, Nature **431**, 162 (2004)
- [29] W. Wootters and W.H. Zurek, *A Single Quantum Cannot be Cloned*, Nature **299**, 802 (1982)
- [30] C. Zalka, *Could Grover’s algorithm help in searching an actual database?*, arXiv:quant-ph/9901068, 1999.

Appendix A: Other Related Algorithms

Push vs Pull

The data-moving map (Definition 3) and parallel look-up map (Definition 2) are both examples of ‘pull’ maps: each ‘instruction’ j_i has a *location* and a *value*; the location describes the *destination* of a data item being transferred (*i.e.* where it’s being pulled to), while the value describes the *source* of that data item (*i.e.* where it’s being pulled from). For completeness, we also describe analogous ‘push’ maps, where the roles of *destination* and *source* are exchanged.

Definition of parallel-push

The ‘push’ analogue of the data-moving map V_N is very trivial to define, since it is nothing other than the inverse of that map, V_N^{-1} . However, the ‘push’ analogue of the parallel look-up map $U_{(N,N)}$ is much more interesting. Since during a parallel operation it is possible that more than one data store might want to ‘push’ data to a given location, any map for a generic pushing algorithm can only be defined with respect to some methodology that describes how data should be combined when such ‘collisions’ occur. A particularly natural way of doing this would be to employ a *computable monoid*. A monoid comes equipped with an associative *group multiplication* operator that allows two (or more) data items to be combined into one. A nice example of a monoid (in fact a group) would be the group of length- d bit-strings with the *xor* operator for group multiplication. (The reader may observe that we did already use this monoid—albeit in a relatively benign fashion—at one point in Definition 2 for the parallel look-up algorithm, for combining target variables, y , with memory data, x .) But other computable monoids can equally well be employed in the definition.

As before, we use N index registers ($\lceil \log_2 N \rceil$ bits each), N target registers (d bits each), and N data registers (d bits each), letting \oplus denote some monoid operator acting on data items (d -bit strings), define the *parallel-push* map as follows.

Definition 4. A logical unitary for parallel push with respect to monoid operator \oplus is a map $W_{(N,N,\oplus)}$ acting on a series of quantum registers that implements

$$\begin{aligned} W_{(N,N,\oplus)} &: |j_1, \dots, j_N\rangle |y_1, \dots, y_N\rangle |x_1, \dots, x_N\rangle \\ &\mapsto |j_1, \dots, j_N\rangle |y_1 \oplus \bigoplus_{j_i=1} x_i, \dots, y_N \oplus \bigoplus_{j_i=N} x_i\rangle |x_0, \dots, x_N\rangle. \end{aligned} \quad (42)$$

When the monoid operator \oplus is non-commutative, the instruction $\bigoplus_{j_i=k} x_i$ should be understood as computing the product of all x_i for which $j_i = k$, with the x_i ordered by lexicographical ascending ordering of the indices i , e.g. $x_1 \oplus x_4 \oplus x_9$.

Parallel push algorithm

The complexity of the parallel push subroutine, $W_{(N,N,\oplus)}$, naturally depends on the complexity of the individual \oplus operation. In all other respects, the complexity is much the same as for the parallel look-up subroutine, $U_{(N,N)}$.

For brevity, we omit a complete description of our algorithm for $W_{(N,N,\oplus)}$, and instead give an overview of how it differs from the algorithm for $U_{(N,N)}$ given in §3.3.

Initial Formatting prepares two packets per index, as before, but this time taking the form $(i, 0, y_i, 0)$ and $(j_i, 1, 0, x_i)$. **Sorting** is the same as before: lexicographic on the address (first entry) then the flag (second entry). At the end of the sort, we are left with a sequence of the form

$$\dots (i, 0, y_i, 0) (i, 1, 0, x_k) (i, 1, 0, x_{k'}) \dots (i, 1, 0, x_{k''}) (i+1, 0, y_{i+1}, 0) \dots \quad (43)$$

The total effect of the **Cascade** will be to load up aux-phase ancillas and to perform monoid operations on the memory-data registers, so that each accumulates the sum of those elements to the right that are in the same ‘ i -block’. **Copying** implements a monoid operation, acting only on the leading packet of each ‘ i -block’, mapping $(i, 0, y_i, x')$ to $(i, 0, y_i \oplus x', x')$, where $x' = \bigoplus_{j_k=i} x_k$, before **Reversing the Cascade**. The **Unsort** step naturally matches the earlier Sort step to reverse it. **Final Formatting** completes the process by copying data out of the packet pairs $(i, 0, y'_i = y_i \oplus \bigoplus_{j_k=i} x_k, 0)$, $(j_i, 1, 0, x_i)$.